COMP 546 project report: Password Cracking

Yahya Hassanzadeh Nazarabadi
yhassanzadeh13@ku.edu.tr

I. MOTIVATIONS

Password Cracking [1] problem has been introduced by the Princeton university in 2003. The Knapsack encryption [2] is a public key encryption [3] scheme. Like all the encryption scheme, the Knapsack encryption consist of three polynomial algorithms: KeyGen, Encryption, Decryption. These algorithms has been described in details in the following subsection. To facilities understanding the scheme, I divided the KeyGen algorithm into two separate algorithms: PubKeyGen and PrivKeyGen generate the public key and secret key respectively.

To encrypt a message, the Encryption algorithm employs the public key on the message and returns the encrypted message. This encrypted message is called the **Ciphertext**. Decrypting the ciphetext with the public key is not efficient and takes a lot of time. In the other word, decrypting the message with the public key is an exponential algorithm. As the size of the public key grows linearly, the time needed to decrypt the ciphertext with the public key grows exponentially.

To decrypt the ciphertext and obtain the message, the Decryption algorithm should be ran on the ciphertext by employing the secret key. Using the secret key, the Decryption algorithm could decrypt the ciphertext and obtain a message in the polynomial amount of time. This justifies the name of the keys; public key is used only for the encryption, while decryption is hard using that. Therefore it could be public and even the adversary could have and use that! However, the secret key should be secret since it could easily decrypt the message.

A. KeyGen: PubKeyGen and PrivKeyGen

1) PrivKeyGen: To encrypt a n bit message, we need n numbers. Without loss of generality assume that we will choose n numbers at random such that the set of that n numbers is super increasingly [4]. In a super increasing set, each element is greater than the sum of all previous elements. In order to encrypt some 6 bits messgae, assume that we chose the super increasing set 1, 2, 4, 9, 20, 38 as the secret key. In this set, each element is greater than the sum of all the previous elements. For example $1 + 2 < 4$ and $1 + 2 + 4 + 9 < 20$.

B. PubKeyGen

To generate the public key, we will multiply all the elements in the secret key set by the number $amodulob$. To select the numbers a and b we have to consider tow points: First, the modules b should be greater than any element in the secret key and therefore should be greater than the sum of all the elements of the secret key. Second, the multiplier a and modules b should not have any factor in common. In the other word $gcd(a, b) = 1$. In our example, we choose the $a = 31$ and $b = 110$. Then the public key is generated by multiplying each element of secret key by a and obtaining the modules b of the result. Having the secret key as $sk = \{1, 2, 4, 9, 20, 38\}$ the public key will be:

$$\begin{aligned} 1 \times 31 &\quad \mathrm{mod}110 = &\quad 31 \\ 2 \times 31 &\quad \mathrm{mod}110 = &\quad 62 \\ 4 \times 31 &\quad \mathrm{mod}110 = &\quad 14 \\ 10 \times 31 &\quad \mathrm{mod}110 = &\quad 90 \\ 20 \times 31 &\quad \mathrm{mod}110 = &\quad 70 \\ 40 \times 31 &\quad \mathrm{mod}110 = &\quad 30 \end{aligned}$$

Therefore the public key is $pk = \{31, 62, 14, 90, 70, 30\}$ and the secret key is $sk = \{1, 2, 4, 10, 20, 40\}$.

C. Encryption

At this point, to encrypt a message, we only consider the messages exactly 6 bits long due to the size of our public key and secret key (each with 6 elements). Assume that we want to encrypt the message 111100. The basic idea of the encryption algorithm is to consider the secret key as a Knapsack with 6 elements. The message is the binary representation of a specific selection of the Knapsack elements. Mapping the message to the secret key set, we will have:

$$1 \times 31 + 1 \times 62 + 1 \times 14 + 1 \times 90 + 0 \times 70 + 0 \times 30 = 197$$

The corresponding ciphertext to the message 111100 (60) would be 197.

D. Decryption

To decrypt a ciphertext, in addition to have the secret key, we have to know numbers a and b that we used to generate the public key at the PubKeyGen algorithm. Since $gcd(a, b) = 1$, a has an inverse modulus b named a^{-1}. In our example, $a = 31$ and therefore $a^{-1} = 71$. Using the a,b which are assumed to be public to everyone and the secret key, one could easily decrypt a ciphertext by multiplying that with $a^{-1} \bmod b$. Therefore to decrypt the ciphertext = 197 we will do:

$$197 \times 71 \bmod 110 = 17$$

Obtaining the result of the multiplication (in this case 17), we will look at the secret key set $sk = \{1, 2, 4, 10, 20, 40\}$ to find the subset that has the sum of 17. This is why we call this encryption scheme **Knapsack** encryption. We have a Knapsack weight limit that obtained by the multiplication and we have a super increasing set (the secret key set). Solving a Knapsack with a super increasing set is very easy ($O(n)$) in a super increasing set. In our example, we start from the last element of the set (40) and pick the elements such that their sum become 17. We could not pick 40 and 20 since their are greater than the Knapsack weight 17, by selecting the whole rest elements $(1, 2, 4, 10)$ we obtain the Knapsack weight of 17. In the other word, we will have:

$$\mathbf{1} \times 1 + \mathbf{1} \times 2 + \mathbf{1} \times 4 + \mathbf{1} \times 10 + \mathbf{0} \times 20 + \mathbf{0} \times 40 = 17$$

The binary representation of this selection would be 111100 which is our original message. The decryption algorithm is therefore terminates correctly.

E. Short Discussion: Hard Knapsack

As considered in the previous section, a Knapsack problem is easy to solve in a super increasing set. Since in such set each element is greater than sum of all previous elements, the subset that has the close some to the Knapsack problem could be easily found. Searching in that subset is much more easier due to the fact that a subset of a super increasing set is itself super increasing. However, in the case of non-super increasing sets, finding such subset is not easy, it takes more time and almost all of the subsets should be examined. Therefore breaking the Knapsack encryption by just having the public key is a hard problem and takes exponential time to examine all the subsets of the public key to find a match to the ciphertext.

II. Problem Definition

The basic nature of this problem is the time/space trade off. We are given the public key of a Knapsack encryption with the size of 25 and we have to break a 25 bits encrypted password to find the original password by examining all the possible subsets of the public key. This operation on the encrypted data is called brutefoce attack [5]. This attack is not efficient, it takes exponential time proportional to the size of the system to break a password.

The next duty is to optimize the bruteforcing attack such that a 50 bits encrypted password could be broken in a reasonable amount of time (under an hour based on the project description [1]). The basic idea for optimization the brute force has been introduced as taking a subset of the public key, computing all possible subset sums that can be made with that subset. Then putting those values into a set and using that set to use that to check all those possibilities with just one lookup [1].

A. Inputs of the problem

The input is a .txt file and an encrypted password. The file contains the public key set and the password has the same amount of bits as the size of the public key set. For example if the public key has 50 elements, password has exactly 50 bits. For the sake of simplicity, the size of each element in the public key set is equal to the size of the password.

Instead of the binary representation for the password and public key set's elements, this project [1] another representation format. Each 5 bits in an element or password represented by a character between a to z or numbers between 0 to 5. The conversion rule is that the decimal value of each 5 bits is calculated. Then it is mapped to a lower case alphabet if the decimal value is less than 27, considering that $a = 0$ and $z = 26$ or to a number between $'0'$ and $'5'$ considering that $'0' = 27$ and $'5' = 32$. For example $(01111)_2 = 15$ therefore it will be mapped to the 'p' character. Therefore 10101000011001001010000010010011001011 1 is converted to the string $vbskbezp$.

B. Output of the problem

The output is **all** the possible strings of characters that the Knapsack encryption of their binary representation with the same public key obtained from the input of the problem yields the same encrypted password obtained from the input.

III. Related Works

A. Brute Force

Implementing the brute force attack [1] is the easiest way to break the Knapsack encryption. It is easy to implement but it takes long time to work. It has been implemented that to break a 25 bits encrypted password. It could break that in a reasonable amount of time, however for 40 bit password (just adding 3 **characters** to the previous system with 25 **bits**). Assuming that it could break the 25 bits password in the $t = T$ seconds using brute force, then as the size of the password increases, t will grows exponentially as follows:

$$\begin{aligned}
&c = 5 && b = 25t = T \\
&c = 6 && b = 30t = T \times 2^5 = 32T \\
&c = 7 && b = 35t = T \times 2^{10} = 1024T \\
&c = 8 && b = 40t = T \times 2^{15} = 32768T \\
&c = 10 && b = 50t = T \times 2^2 5 = 33554432T
\end{aligned}$$

In the above equations, c is the size of encrypted password in characters as described in the Section II. b is the size of the password in the bits and t is the time needed to break that password using bruteforce method. If bruteforce could break the the 25 bits password in 15 seconds, the time needed to break the 40 bits and 50 bits password would be something near to $15 \times 32768 = 491529$ seconds (6 Days) and $15 \times 33554432T = 503316480$ seconds or 15 years!

Brute force needs a few amount of memory. Since it just produces a sum, checks it and outputs the result. Therefore the upper bound for the memory space would be $O(1)$

B. Dynamic programming

The other solution is to model the problem as a zero-one Knapsack. The encrypted password obtained by the input of the problem is the Knapsack weight. In order to decrypt the password, we have to find a subset of the public key elements with summation exactly equal to the Knapsack weight. We already modeled the problem in similar way in the previous sections. The main reason to reconsider that is the glorious solution for the zero-one Knapsack problem based on the dynamic programming [6], [7].

There are two main problems regarding to employ dynamic programming to break the Knapsack encryption. First, the memory and time needed to solve this problem growth exponentially. To break a n bit password with the decimal value of d, we need to generate a table with n rows and d columns, solving all the zero-one Knapsack problems from 0 to d. To obtain an upper bound for the memory space, since the password contains of n bits, the maximum value it could be have is 2^n. Therefore the maximum number of column that the dynamic programming table could have is 2^n. The memory complexity of this approach is therefore $O(n2^n)$. Time complexity of this approach is also bounded by the time needed to create this table and therefore would be $O(n2^n)$.

The second problem is that dynamic programming method, by itself, would not provide all the possible solutions. To achieve that, a backtracking algorithm should be added to traverse the table for all possible solutions.

C. Subset sum problem

Subset sum problem [7] is NP-complete [8]. There are some pseudo-polynomial and enhanced exponential algorithms [9] to solve the problem based on the exact value order of the elements in the set. However, there is a solid restriction in the Knapsack encryption. The size of elements in the public key set is equal with the size of encrypted and original password. Therefore all the summations should be ran modulus $2^{PasswordSize}$. Otherwise, sum of some subsets may go beyond the $PasswordSize$. This makes the enhanced subset sum solutions to be inefficient since their basic idea is the magnitude of the elements and modulus calculation will not assist them in that issue.

IV. Solution: Enhanced Bruteforce

The goal of the project was to optimize the bruteforce method [1]. It asked to implement an algorithm that is functionally like the brute-force but it could break a 50-bits (10 chars) encrypted password in a reasonable amount of time (under an hour [1]). To do so, the project description suggested to find a subset of the public key, calculate all the possible subset sums of that subset. Use that calculated subset sums to break the password.

Algorithm IV.1 shows the proposed solution to enhance the brute-force mechanism. The algorithm receives the public key (PK) file and the encrypted password (EncPass) in the format described in section II as its inputs. It first divides the whole PK file set into two sets with almost the same size: $subTable1$ and $subTable2$ (Algorithm IV.1, line 1). It then calculates all the possible subset sums of the $subTable2$ and stores them in $subResult$(Algorithm IV.1, line 2). The decimal value of the encrypted password (DecPass) will be calculated at the next step (Algorithm IV.1, line 3).

At this moment, the critical part of the algorithm will be started, it will create every possible subset sum of the $subTable1$ and the corresponding value that is needed from $subResult$ that their summation would obtain the DecPass. It names that corresponding value as the ST (Algorithm IV.1, line 5). It then searches the ST in the $subResult$ using a Binary Search [7] (Algorithm IV.1, line 6). If ST was found in the $subResult$, the algorithm calculates the binary representation of a subset selection from $subTable1$ and $subResult$ such that the sum of elements in that subset will result the EncPass (Algorithm IV.1, line 7). It then encode that binary representation using the representation format described in the section II and print the encoded result as the original password (decryption of the EncPass) (Algorithm IV.1, line 5).

Algorithm IV.1: Knapsack Craking

Input: file PK, string EncPass
Output:

```
1 (subTable1,subTable2) = split(PK);
2 subResult = AllSubsetSum(subTable2);
3 DecPass = Parse(EncPass);
4 for i=0; i < 2^subTable1.size(); i++ do
5     ST = (DecPass − subTable1.i^th subSetSum) mod
      2^EncPass.size()
6     if BinarySearch(subResult, ST) then
7         BinRep = subset binary representation;
8         print(encode(BinRep));
```

The dominating part of the proposed algorithm is the loop part (Algorithm IV.1, line 4). For a n bit password that loop will be iterated $2^{\frac{n}{2}}$ times. Since it calculates all the subset sums of the first half of the public key set. In each iteration, the algorithm searches for the ST in the $subResult$ list with the size of about $2^{\frac{n}{2}}$ elements(since it contains the all possible subset sum of second half elements of the list). A binary search in a list with $2^{\frac{n}{2}}$ has the cost of $O(\log 2^{\frac{n}{2}})$ which is equal to $O(n)$. Therefore the runnig time of the proposed algorithm is $O(n2^{\frac{n}{2}})$.

V. Implementation Issues

A. Data Structures

$subTable1$, $subTable2$ and $subResult$ have been implemented by Java Arraylists [10]. The main reason to choose Arraylist instead of a link list was the insertion and retrieval time of Java Arraylists. In the Java Arraylists class, by defining the upper bound of the size, the insertion and retrieval time complexity for an element would be $O(1)$. In the case of the link lists, although the insertion time is $O(1)$, retrieving an object in a certain index takes $O(n)$ due to the need of list linear traversal. In comparison to the ordinary arrays, Java array list supports lots of auxiliary operations such as sorting

Algorithm	Time	Space	Find All
BruteForce	$O(2^n)$	$O(1)$	Yes
Dynamic Programmign	$O(n2^n)$	$O(n2^n)$	No
Enhanced Bruteforce	$O(n2^{\frac{n}{2}})$	$O(2^{\frac{n}{2}})$	Yes

TABLE I
COMPARISON BETWEEN VARIOUS METHODS CRACKING THE KNAPSACK ENCRYPTION

based on one data member of an object, shuffling an Arraylist etc.

All the Arraylists described above contains the instance objects of the type $symbol$. The $symbol$ data type has been created by the author of this report to manage the subset sums and their indexes. Each $symbol$ object has two parts: $decRep$ which represents the decimal value of the symbols and $index$ which represents the subset of the public key elements that their sum is equal to the $decRep$. When the public key is loaded into $subTable1$ and $subTable2$, the $index$ of each element is initiated with the index of that element in the public key file. Whenever two symbols are added together their indexes will be combined. Therefore at each moment the index of each symbol element reflects the subset of the public key file elements that their sum is equal to the $decRep$ of that index. At this point, this question may come to the mind that when two symbols will added together? The answer is that when we want to find all possible subset sums of a set ($subTable1$ or $subTable2$). During that procedure symbols are added together repeatedly.

B. Binary Search

In the Algorithm IV.1, in order to implement the binary search operation, the binary search from the Java Collections Platform [11] has been employed. In order to use the binary search, the list should be sorted. For the sorting algorithm, the merge sore algorithm from the Java Collections Platform was used. The reasons to choose this algorithm was its $O(n \log n)$ performance in the average average and worst case, its Java Collections Platform built in feature which makes it fully compatible for the Arraylist types and finally in comparison to the quick sort algorithm it did not required prepossessing the list (shuffling for example). However, the worst case memory complexity of the merge sort was O(n) but since I only was used once during the whole execution of the algorithm, this worst case space usage in comparison to the $O(2^{\frac{n}{2}})$ space needed for breaking the Knapsack was negligible.

VI. COMPARISON WITH THE RELATED WORKS

Table I shows the comparison between the proposed solutions and other best known methods. The *Find All* feature stands for the case that whether the algorithm will find all the possible solutions or not. In the other word, since the Knapsack encryption is not collision resistance, several passwords may be mapped to one ciphertext. In this situation, some algorithms like the dynamic programming solution would not find all the possible solutions by themselves and need additional algorithms as was considered in the section III. As it is shown in the table I, by making a trade-off between the time and space, the Enhance Brutefoce could break the Knapsack Encryption in a significantly better time.

Just to make a practical sense, as was considered in the section III, if the time needed to break a 25 bits encrypted password with the enhanced brute force algorithm is $t = 2^{25} = T$, the time needed to break a 50 bits encrypted password with the enhanced brute force method is $25 \times 2^{25} = 25 \times T$.

VII. CONCLUSION

The main goal of this project was to break the Knapsack Encryption. This problem is a variation of the subset sum problem which is NP-complete. We observed that however we could not provide a polynomial solution to this problem, but we could enhance the current algorithms to provide a feasible solution for a certain problem size.

The main task of the problem was to enhance the brute force algorithm such that it could break the 50 bits (10 chars) password in a reasonable amount of time (under an hour). By the proposed solution, with a Dell Latitude E6330, the author could break the 50 bits encrypted password in about 3 minutes.

The problem shows a very spectacular trade off between the time and space. Using the pure brute force takes a lot of time and even days. With giving help from the memory and increasing the memory complexity, in the enhanced bruteforce, we could decrease the time complexity significantly. Breaking the 50 bits encrypted password with the Enhanced Brute force method takes only **3 minutes** while it takes about **15 years** in the case of the ordinary brute force!!

REFERENCES

[1] P. university. (2003) Password cracking. [Online]. Available: http://www.cs.princeton.edu/courses/archive/spring03/cs226/assignments/password.html

[2] R. Merkle and M. E. Hellman, "Hiding information and signatures in trapdoor knapsacks," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 525–530, 1978.

[3] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2014.

[4] Wikipedia. Super increasing set. [Online]. Available: http://en.wikipedia.org/wiki/Superincreasing_sequence

[5] Bruteforce attack. [Online]. Available: http://en.wikipedia.org/wiki/Brute-force_attack

[6] A. Skorkin. Algorithms, a dropbox challenge and dynamic programming. [Online]. Available: http://www.skorks.com/2011/02/algorithms-a-dropbox-challenge-and-dynamic-programming/

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

[8] Np-complete. [Online]. Available: http://en.wikipedia.org/wiki/NP-complete

[9] Subset sum problem. [Online]. Available: http://en.wikipedia.org/wiki/Subset_sum_problem

[10] Oracle. Class arraylist. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

[11] ——. Collections java platform. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html